GPU Code Optimization using Abstract Kernel Emulation and Sensitivity Analysis

Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, Prashant Singh Rawat, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, Ponnuswamy Sadayappan

GPU Code Optimization using Abstract Kernel Emulation and Sensitivity Analysis

CHANGWAN HONG, The Ohio State University, USA
ARAVIND SUKUMARAN-RAJAM, The Ohio State University, USA
JINSUNG KIM, The Ohio State University, USA
PRASHANT SINGH RAWAT, The Ohio State University, USA
SRIRAM KRISHNAMOORTHY, Pacific Northwest National Laboratory, USA
LOUIS-NOËL POUCHET, Colorado State University, USA
FABRICE RASTELLO, Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France
P. SADAYAPPAN, The Ohio State University, USA

In this paper, we develop an approach to GPU kernel optimization by focusing on identification of bottleneck resources and determining optimization parameters that can alleviate the bottleneck. Performance modeling for GPUs is done by abstract kernel emulation along with latency/gap modeling of resources. Sensitivity analysis with respect to resource latency/gap parameters is used to predict the bottleneck resource for a given kernel's execution. The utility of the bottleneck analysis is demonstrated in two contexts: 1) Coupling the new bottleneck-driven optimization strategy with the OpenTuner auto-tuner: experimental results on all kernels from the Rodinia suite and GPU tensor contraction kernels from the NWChem computational chemistry suite demonstrate effectiveness. 2) Manual code optimization: two case studies illustrate the use of the bottleneck analysis to iteratively improve the performance of code from state-of-the-art domain-specific code generators.

CCS Concepts: • **Computing methodologies → Modeling methodologies**;

Additional Key Words and Phrases: Performance modeling, GPU, abstract emulation, bottleneck analysis, sensitivity analysis

Authors' addresses: Changwan Hong, The Ohio State University, Columbus, OH, USA, hong.589@osu.edu; Aravind Sukumaran-Rajam, The Ohio State University, Columbus, OH, USA, sukumaranrajam.1@osu.edu; Jinsung Kim, The Ohio State University, Columbus, OH, USA, kim.4232@osu.edu; Prashant Singh Rawat, The Ohio State University, Columbus, OH, USA, rawat.15@osu.edu; Sriram Krishnamoorthy, Pacific Northwest National Laboratory, Richland, WA, USA, sriram@pnnl.gov; Louis-Noël Pouchet, Colorado State University, Fort Collins, CO, USA, pouchet@colostate.edu; Fabrice Rastello, Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France, fabrice.rastello@inria.fr; P. Sadayappan, The Ohio State University, Columbus, OH, USA, sadayappan.1@osu.edu.

Manuscript submitted to ACM

P. SADAYAPPAN, The Ohio State University, USA

ACM Reference Format:
Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, Prashant Singh Rawat, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, and P. Sadayappan. 2018. GPU Code Optimization using Abstract Kernel Emulation and Sensitivity Analysis. 1, 1 (December 2018), 37 pages. https://doi.org/10.1145/3192366.3192397

1 INTRODUCTION

Code transformations for optimization are typically guided by performance models. However, two significant challenges are faced by optimizing compilers for GPUs:
1) The complexity of modeling the multiple concurrent interacting hardware components in a GPU makes it extremely challenging to develop sufficiently accurate performance models that can reliably predict which of two alternative code structures will execute faster on a given GPU; and
2) Even if a sufficiently discriminating performance model is developed, the space of semantically equivalent code structures for most compute-intensive algorithms is extremely large when considering the number of possible ways of mapping the statement instances to threads/thread-blocks.

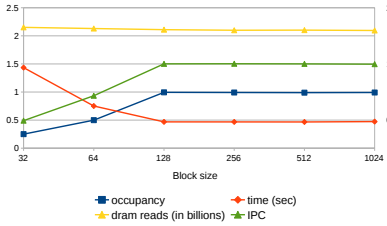

Fig. 1. Grid reshaping

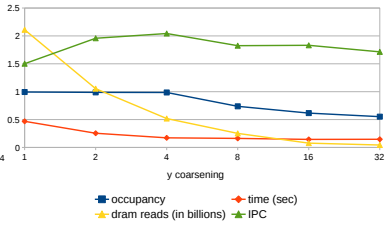

Fig. 2. Thread coarsening

```
int tx = threadIdx.x, i = blockIdx.x*32 + tx,
    j = blockIdx.y;
__shared__ float cb[32];
float sum = 0.0;
for( int ks = 0; ks < p; ks += 32 ){
  cb[tx] = c[ks+tx+pitch_c*j];
  for( int k = ks; k < ks+32; ++k )
    sum += b[i+pitch_b*k] * cb[k-ks];
}
a[i+pitch_a*j] = sum;
```

Listing 1. Matrix Multiplication

Production compilers therefore use simple and fairly imprecise cost models to guide optimizing transformations, in part because very precise performance models usable in compilers are unavailable, and in part because production compilers cannot afford excessively high compile times in performing an extensive search over a large configuration

space. But application developers are willing to wait for minutes or even hours for the "final" compilation of a production code, if the performance of the resulting compiled code can be significantly improved by a slow but effective optimizing compiler. Production compilers do not address this use case, but this is the scenario we target in this work.

In this paper, we present a new approach to GPU code optimization with the following features:

- Code optimization is driven by the identification of bottleneck hardware resources.
- Instead of using analytical performance models, performance modeling of GPU kernel execution is done via fast abstract kernel emulation, along with simplified modeling of two key hardware parameters for resources in the GPU: latency and gap (inverse throughput).
- A novel sensitivity analysis with respect to hardware resource parameters is used to identify resource bottlenecks.
- Experimental evaluation of performance modeling is done using all kernels of the Rodinia benchmark suite.
- Utility in code optimization is demonstrated in two ways: i) automated GPU kernel optimization by coupling bottleneck-analysis driven search with auto-tuning; ii) two manual case studies of improving the performance of code generated by domain-specific code generators for tensor contractions [19] and stencil computations [27].

2 OVERVIEW OF APPROACH

In this section, we use examples to present an overview of the new approach to performance modeling and bottleneck identification for GPU kernel execution. We first illustrate some of the key factors influencing GPU kernel performance and the impact of bottleneck resources.

2.1 Bottleneck Resources on GPUs

The performance of a GPU kernel is typically constrained by one of its resources, such as the global memory subsystem, shared memory, or the special function unit. We begin with a simple illustration of the limiting bottleneck resource and how kernel performance optimization can be guided by the knowledge of the limiting bottleneck.

Performance under limited concurrency: Fig. 1 shows the execution time of a CUDA kernel code for dense matrix-matrix multiplication. The parallel i and j loops of the standard triple-nested loop matrix multiplication code are mapped to threads in the grid, and the innermost loop over k performs a dot product. It operates on 2048×2048 matrices, using 1D thread-blocks, on an NVIDIA K20c GPU. The figure shows occupancy, defined as the ratio of active warps on an SM (Streaming Multiprocessor) to the maximum number of active warps the SM can support. In addition, it reports DRAM accesses, instructions per cycle (IPC), and execution time with varying thread block size. We observe the performance improvements follow the occupancy closely. For a thread block size of 128, an occupancy of 1 is achieved, delivering the best performance. Since the number of DRAM accesses remains constant for all cases, increasing occupancy helps tolerate the latency of global memory accesses. In this regime (thread block size < 128), the global memory is the resource bottleneck, and performance is limited by memory access latency. As occupancy increases, the code is better able to tolerate global memory latency and performance improves until maximum occupancy is achieved. Further increase in thread-block size does not result in any further increase in concurrency and there is no improvement in performance.

Latency-bound vs. throughput-bound resource: Beyond a thread-block size of 128, performance is limited by global memory bandwidth, not by global memory latency. Thus the same hardware resource can be the performance limiting bottleneck in different ways: latency-bound or throughput-bound. When a resource is latency-bound, increasing the number of concurrent requests to that resource can improve performance. When a bottleneck resource is throughput-bound, the only way to improve performance is to reduce the total demand on the resource, as shown below.

Enhancing data reuse by thread coarsening: Thread coarsening [21, 31] is an approach to achieving register tiling for GPU kernel code. Fig. 2 shows the performance trends for a thread-coarsened version of the matrix-matrix multiplication code. The total number of global memory loads gets reduced. because the number of thread blocks along y are halved, and each thread performs the same number of global memory load operations. For a thread-coarsening factor of 2, halving the volume of DRAM transactions leads to proportional improvements in the execution time. Execution time and IPC continue to

improve until a thread coarsening factor of 4. While thread coarsening increases the per-thread register use, this does not impact the occupancy for coarsening factor less than 4. Beyond a coarsening factor of 4, the hardware limit on available registers per SM causes the number of active loaded warps (and thus occupancy) to decrease. At this point, the kernel's performance is once again bound by memory latency. Due to this limitation, further reductions in the volume of DRAM loads do not improve performance.

2.2 SAAKE

In this paper, we present SAAKE(Sensitivity Analysis via Abstract Kernel Emulation), an approach to identify the bottleneck resource for a given program version and whether the resource is latency or throughput bound. The objective is to identify the bottleneck resources for a given kernel binary (SASS) code. To this end, we employ sensitivity analysis, i.e., we evaluate the potential performance impact of changing the modeled latency or throughput parameters of a resource. This approach requires performance modeling, motivating our development of a lightweight kernel emulation approach.

Abstract Kernel Emulation: In contrast to analytical performance modeling, we perform an abstract emulation of the actual kernel binary (SASS) code in an emulator. Only a (tiny) fraction of the thread blocks to be executed are emulated, typically taking only a few milliseconds. During abstract kernel emulation, ready instructions from active warps are scheduled for execution using some warp scheduling strategy, such as Greedy Then Oldest (GTO) [28].

Each primary hardware resource in the GPU is modeled using two parameters: *latency* and *gap*. The *latency* of a resource is the total time for a request to be completed by that resource from start to finish. For pipelined hardware resources, multiple concurrent requests can be in flight, but a new request may not be processable every clock cycle. The minimum number of cycles between the start (or the end) of execution of two successive requests at a resource is its *gap*. Thus, the maximum throughput achievable by a resource is one request per *gap* cycles, i.e., the gap is inversely related to the peak throughput of the resource.

The completion time of an instruction is modeled by adding the latency of the needed resources (e.g., shared-memory, global-memory, special functional units, etc.). For each resource, an "earliest admission" time is maintained, which is incremented by its gap

when a new request starts execution on it. Dependences between instructions are tracked, so that the earliest schedulable time for an instruction is later than the completion times of all previously scheduled instructions it depends on.

Bottleneck Identification: To detect resource bottlenecks, we use sensitivity analysis with respect to resource model parameters. This is done by performing multiple kernel emulations using modified resource parameters for latency and gap. First kernel execution time is modeled using resource model parameters that correspond to the target GPU, determined via use of microbenchmarks. Next, kernel emulation is repeated with one resource parameter changed, say increase modeled latency of global memory by 10%. Kernel emulation is then performed by changing the global memory gap parameter by 10%. Similarly, emulations with modified parameters for each of the hardware resources is performed, with one resource parameter modified for each emulation. The hardware resource with the largest relative change in predicted kernel execution time (over the base time) is identified as the bottleneck resource. Further, whether the large change occurred with the change of latency or gap parameter points to whether the resource is latency bound or bandwidth bound.

Sometimes, no single resource parameter may stand out as the clear bottleneck, with multiple resource parameters exhibiting comparable and moderately high sensitivity. In this case, it is possible that different portions of a GPU kernel are constrained by different bottlenecks. The sensitivity analysis can then be performed over different portions of the kernel, as illustrated later through a case study on optimizing a tensor contraction kernel.

3 BOTTLENECK IDENTIFICATION VIA SENSITIVITY ANALYSIS

In this section, we present the key idea behind the proposed approach to GPU kernel optimization. To illustrate the technique, we restrict ourselves to a simple analytical performance model for a *single* pipelined hardware resource such as an arithmetic functional unit in one of the SMs of a GPU. More realistic scenarios of complex kernels using the multiple asynchronous pipelined units are discussed in the next section.

3.1 Resource Parameters

A pipelined hardware resource is modeled by two fundamental performance parameters: *latency* and *gap*. The *latency* (denoted L) is defined as the number of cycles an operation waits before execution, when it depends on the immediately preceding operation[1]. When two operations depend on one another, their starting time has to be separated by the latency of the first. The *throughput* is the number of operations a resource can issue (equivalently complete) per cycle. The *gap* (denoted G) is the inverse of the throughput. Because the resource is pipelined, its *gap* is lower than its *latency*.

3.2 Modeling Performance Impact of Concurrency

The total *overall concurrency* (denoted C) in a warp's execution is the product of warp-level parallelism (WLP) and instruction-level parallelism (ILP) within each warp. Consider a repetitive loop, where instructions of the current iteration only depend on results computed by the corresponding instruction in the previous iteration. The number of iterations in the loop is denoted by P, the *number of phases*.

For this simple scenario, the total execution time can be modeled as shown in Fig. 3. There are two cases:

Latency-limited execution ($L > C \times G$): Fig. 3 (left) depicts the execution in this case. In the first phase, successive instructions can only be issued once every G cycles since they all need the same pipelined hardware resource. Since $L > C \times G$, the first instruction of each phase phase can be issued L cycles apart. The entire execution is completed after $T = L \times P + (C - 1) \times G$ cycles.

Throughput-limited execution ($L \leq C \times G$): Fig. 3 (right) illustrates the scheduling of operations in this case. Although the first instruction of the second phase has its input operand ready after L clock cycles, (satisfying the dependence on the corresponding instruction from the first phase), the gap constraint means that it cannot be issued until $C \times G$ cycles after the first instruction of phase one. The total completion in this case is $T = L + (C \times P - 1) \times G$ cycles.

[1]Typically, latency is defined as the number of cycles for an operation to complete. We use a modified definition to account for architectural features such as pipeline forwarding.

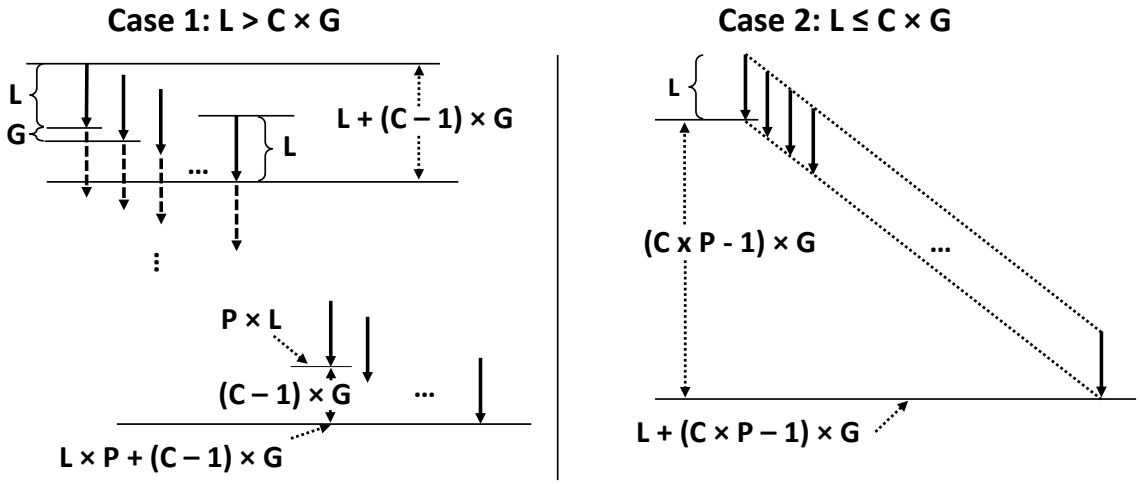

Fig. 3. Illustration of latency-limited (left) and throughput-limited (right) execution

3.3 Sensitivity Analysis for Bottleneck Identification

In this work, we perform sensitivity analysis by observing the fractional increase/decrease of execution time ($\frac{\Delta T}{T}$) for a given fractional increase/decrease of a resource parameter ($\frac{\Delta L}{L}$ for *latency* and $\frac{\Delta G}{G}$ for *throughput*) - equivalently, comparing $\frac{\Delta T}{\Delta L} \times \frac{L}{T}$ (resp. $\frac{\Delta T}{\Delta G} \times \frac{G}{T}$) with 0. We use ΔX notation here to reflect the discrete property of resources. However, a continuous form of the analytical expressions for our example can be obtained as follows:

	$\frac{\delta T}{\delta L} \times \frac{L}{T}$	$\frac{\delta T}{\delta G} \times \frac{G}{T}$
Latency-Limited	$1/\left(1 + \frac{(C-1)G}{LP}\right)$	$1/\left(1 + \frac{LP}{(C-1)G}\right)$
Throughput-Limited	$1/\left(1 + \frac{(CP-1)G}{L}\right)$	$1/\left(1 + \frac{L}{(CP-1)G}\right)$

Consider the *latency*-limited case. As the larger L is compared to $C \times G$, the sensitivity expressions ($\frac{\delta T}{\delta L} \times \frac{L}{T}$) (based on *latency*) and $\frac{\delta T}{\delta G} \times \frac{G}{T}$ (based on *gap*) become 1 and 0, respectively. We can derive symmetric conclusions for the *gap*-limited case, leading to the following rules:

	$\frac{\delta T}{\delta L} \times \frac{L}{T}$	$\frac{\delta T}{\delta G} \times \frac{G}{T}$
Latency-limited	$\gg 0$	≈ 0
Throughput-limited	≈ 0	$\gg 0$

We use the above observations to identify the mode of bottleneck (latency vs. throughput). For more complex kernels and multiple GPU resources, accurate analytical modeling to identify bottleneck resources is extremely challenging. However, sensitivity analysis with respect to each resource is feasible by performing multiple abstract kernel emulations with changed resource parameters, as described in the next section. When analyzing sensitivity of multiple resources, non-bottleneck resources will show little change in sensitivity with respect to both *latency* and *gap*. The resource exhibiting the largest sensitivity analysis metrics is identified as the bottleneck resource.

4 ABSTRACT KERNEL EMULATION

This section details the approach to abstract kernel emulation. The algorithm uses the *latency* and *gap* parameters for each resource in the target GPU architecture. Due to space constraints, we do not discuss how these parameters are obtained; a technical report [12] provides some details. Our approach is similar to the one described by Papadopolou [25].

4.1 Overview

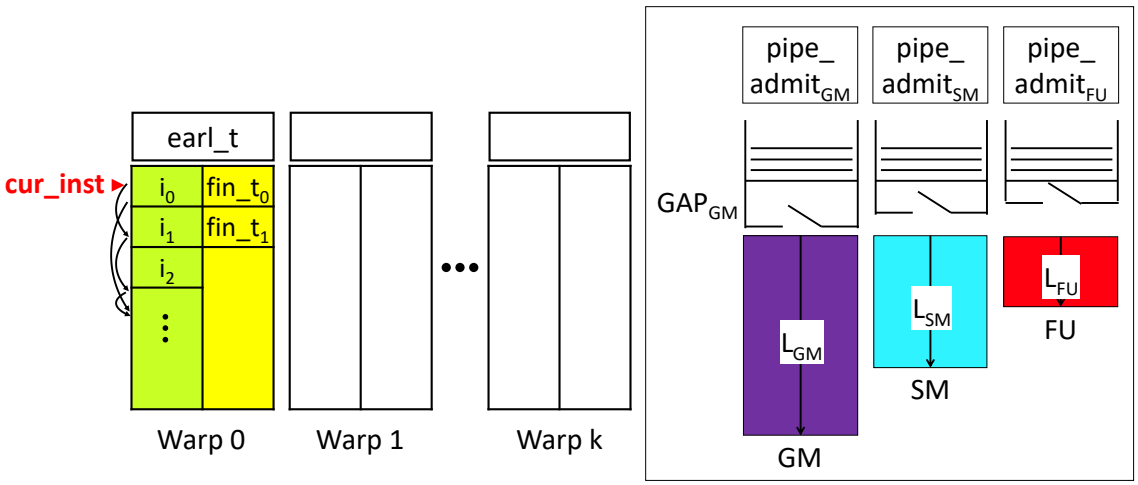

Fig. 4. Overview of abstract kernel emulation

Fig. 4 illustrates the approach to kernel emulation. A maximal number of warps that can concurrently occupy one SM of the GPU get modeled. Each warp is modeled by

a current-instruction pointer and an earliest-schedule-time for the current instruction. Each modeled hardware resource is associated with a latency and gap parameter. Each resource is essentially treated as having an unbounded input queue for requests, with one waiting request being allowed to enter the resource every *gap* units of time. The state of each hardware resource is maintained by a *pipe_admit* time, which represents the earliest time at which a new request to that resource can enter that resource pipeline. The current instruction of a warp is eligible to be scheduled if all needed operands are ready. When an instruction is scheduled, the *pipe_admit* of the needed hardware resource is the time at which the processing of the operation will begin. Adding the *latency* of the resource determines when that instruction will complete, which is recorded in the *fin_t* entry associated with that instruction. Within each warp, intra-instruction dependences are tracked. When a new instruction from some warp is scheduled, the availability status of its operands at that time is known since the finish-times of all preceding instructions have been recorded in the *fin_t* entries of the producer instructions.

4.2 Illustrative Example

Consider an SM with the following resource parameters: global-memory latency = 500; global memory gap = 100; functional unit latency = 100; functional unit gap = 20. Figure 5 shows different stages of abstract kernel emulation with 3 warps. Initially, statement $S11$ in the first warp reads a value from global-memory into a register. Since $S11$ is not dependent on any other instruction and the global-memory is ready to accept/admit a request, the statement can be issued immediately. Once $S11$ is issued, global-memory requests are blocked for 100 cycles (the *gap* parameter of the global-memory resource). $S11$ will require 500 cycles (global-memory *latency*) to finish its execution. The next instruction, $S12$, is an add instruction requiring the functional unit. Because $S12$ does not depend on any previous instruction and the functional unit is idle, it can be scheduled at clock cycle 1. $S12$ will require 100 cycles to finish execution. The next instruction of warp 1, $S13$, is dependent on $S12$. Since $S12$ will only be completed at clock cycle 101, $S13$ cannot be scheduled immediately and we move to warp 2. $S21$, $S22$, $S31$, and $S32$ follow the same pattern as $S11$ and $S12$. $S32$ is scheduled in clock cycle 4 and completes execution in clock cycle 141. At clock cycle 5, no warp has an instruction that can be scheduled. The

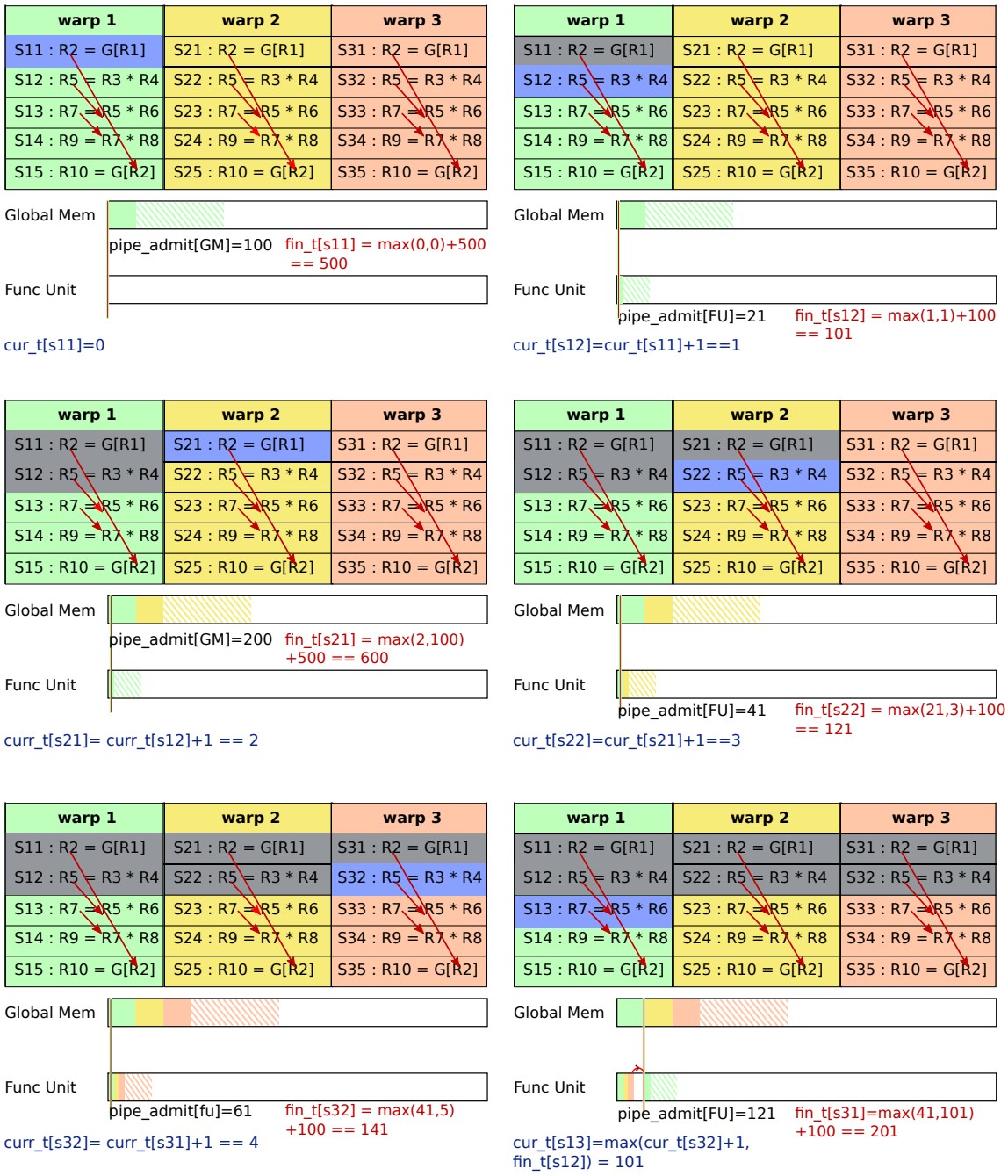

Fig. 5. Illustration of abstract kernel emulation

next instruction that can be scheduled is $S13$ at clock cycle 101. The clock is advanced to step 101 and $S13$ is selected for scheduling.

The abstract kernel emulation is based on a Greedy-Then-Oldest (GTO) scheduling policy [28]. We use GTO because the actual warp scheduling policy used in Nvidia GPUs is not publicly known. Alternative scheduling policies can easily be incorporated into the emulator.

4.3 Emulation Algorithm

Alg. 1 outlines the abstract emulation algorithm to predict kernel execution time. Several details, such as handling of conditional statements, L1/L2 cache misses, and uncoalesced global-memory access, are not included in this high-level pseudocode; these issues are discussed at the end of this sub-section. Modern GPUs have multiple warp schedulers [35, 36] and each warp scheduler can issue more than one independent instruction from the same warp in a single clock cycle. For simplicity, Alg. 1 models a single warp scheduler that schedules a single instruction per cycle. However, the implementation of the abstract kernel emulator models the actual number of warp schedulers and the number of instructions scheduled per cycle per warp scheduler.

Abstract kernel emulation begins by scheduling the first instruction from warp 0 and proceeds until all instructions from all warps are scheduled. At each step, a warp with a ready instruction is selected. For a warp's current instruction (*cur_inst*), the earliest scheduling time (*earl_t*) is determined as the maximum among the finish times of its predecessors and the current clock (lines 5-7). If the current instruction cannot be scheduled at the current clock, then the warp whose instruction has the smallest earliest_schedule time is sought (line 9) and the clock is advanced (line 10). If the current instruction can be scheduled in the current clock cycle, it is scheduled for execution. A scheduled instruction may not execute immediately, but might be enqueued if the needed hardware resource is not available. Thus, the actual time at which an instruction begins execution is the maximum of the current clock and the clock cycle at which the corresponding resource r is ready to accept a request (*pipe_admit*). An instruction's finish time (*fin_t*) is the sum of the time at which its execution begins and the corresponding resource's latency. Once a resource accepts a request, it is unavailable to accept another request for *gap* cycles: The corresponding resource's pipe_admit time is updated by adding *gap* (line 14). On line 15, *start* represents the time at which the previous instruction (the one

Algorithm 1: Abstract kernel emulation

input : Number of warps in one block, number of warps in one SM, *latency* and *gap* of each resource
output : Estimate of execution time, utilization of resources

1 ***t*: current time, *w*: current warp, *r*: current resource, *i*: current instruction**
2 **fin_t**$[w][i]$: finish time of instruction i in warp w
3 **cur_inst**$[w]$: current instruction of warp w (initialized to 0)
4 **pipe_admit**$[r]$: earliest time when the next instruction can be admitted to the pipeline of resource r (initialized to $-\infty$)
5 **start**$[r]$: time at which the last instruction that uses r was issued
6 **utilization**$[r]$: total time resource r was active (initialized to 0)
7 **latency**$[r]$: latency of resource r, **gap**$[r]$: gap of resource r
8 $p \rightarrow q$: instruction q is dependent on instruction p
9 **earl_t**$[w]$: earliest schedule time of the current instruction of warp w (initialized to 0)
10 **end**: non executable last instruction of each warp

11 $t \leftarrow 0$
12 $w \leftarrow$ warp 0
13 **while** $\exists x$ ***s.t.*** $cur\_inst[x] \neq end$ **do**
14 $i \leftarrow$ cur_inst$[w]$
15 **if** $i \neq end$ **then**
16 earl_t$[w] \leftarrow \max(\{\text{fin_t}[w][p] \mid p \rightarrow i\},\ t)$
17 **else** earl_t$[w] \leftarrow \infty$;
18 **if** $earl\_t[w] > t$ **then**
19 **find** w **s.t.** earl_t$[w] = \min_x$ earl_t$[x]$
20 $t \leftarrow$ earl_t$[w]$
21 **continue**
22 $r \leftarrow$ resource(i)
23 fin_t$[w][i] \leftarrow \max(t, \text{pipe_admit}[r]) + \text{latency}[r]$
24 pipe_admit$[r] \leftarrow \max(\text{pipe_admit}[r], t) + \text{gap}[r]$
25 utilization$[r] \leftarrow \text{utilization}[r] + \min(t - \text{start}[r],\ \text{latency}[r])$
26 start$[r] \leftarrow t$
27 cur_inst$[w] \leftarrow$ cur_inst$[w] + 1$
28 $t \leftarrow t + 1$
29 **return** $(\max_w$ fin_t$[w][\text{cur_inst}[w] - 1])$, utilization$[]$

before i) got issued on resource r. If *latency* is smaller than elapsed time in ($t - start$), it means r was idle during the remaining time ($t - start - latency$). The utilization of resource r is updated by adding only the time during which it was active (line 15). Finally, the instruction pointer of the current warp is updated (line 17) and the clock is updated

(line 18). This process is repeated until all instructions from all warps are scheduled. After all the instructions are scheduled, the clock is set to the latest finish time of the last completing instruction among all warps (argument 1 of line 19).

The kernel emulation models the execution of a maximal set M (which is determined based on the resource requirements of each thread-block) of thread-blocks that can concurrently execute on an SM. The number of needed "phases" to execute all thread-blocks of a kernel is modeled by dividing the total number of thread-blocks by the product of M and the number of SMs in the GPU. Thus, the total emulation time is generally a small fraction of the actual execution time of compute-intensive kernels.

4.4 Additional Details

We now present some additional details regarding performance modeling via abstract kernel emulation that were omitted in Alg. 1.

Conditional Statements: For if-then-else conditions and loop bounds that are dependent on known parameters, emulated instructions reflect actual executed instructions [34]; for loops with unresolvable loop bounds, representative trip counts are provided to the emulator; for if-then-else conditions dependent on dynamically computed values, all predicated control paths are conservatively emulated in lexical order of SASS code.

Irregular/uncoalesced data access: These can be identified by static analysis but was manually identified since a static analysis for it has not been implemented. A full DRAM transaction request is conservatively assumed to occur for each load/store from every thread in a warp for uncoalesced accesses. In contrast, for coalesced accesses only one DRAM transaction is scheduled for the entire warp.

L1/L2 cache: The cache is not explicitly emulated. However, modeling accuracy can be enhanced by providing the emulator an optional cache miss-rate parameter for a region of the code. Although no cache simulation is performed in the emulator, the availability of actual or estimated cache miss ratio is used in the following manner in the emulation. An additional L2 cache resource is added to the abstract emulation. For every emulated global memory access instruction, its execution is modeled by randomly assigning it to either the modeled L2 cache resource (with lower gap and latency parameters) or the global memory resource (with higher gap and latency parameters). The probability of

assigning the execution of the load to the modeled L2 resource is the provided cache miss rate. We present experimental results in the next subsection that demonstrate the improvement in performance prediction accuracy from such modeling.

Barrier synchronization: Every warp in a block is stalled at barrier synchronization statements until all instructions in the block finish execution.
Bank conflicts: For cases where indices of shared memory are dependent on known parameters, bank conflict can be emulated. Register bank conflicts can also be emulated as described in the literature [15, 40]. Register bank conflicts are sensitive to register names that can be extracted from the SASS codes. On the Kepler GPU, there are four register banks for each thread [15, 40] and operands can read only one value from each register bank per cycle.
Atomic operations: Similar to the handling of if statements and bank conflict, if indices of array atomic operations are dependent only on known parameters, atomic operations can be emulated.
Instruction queue: Instruction queue lengths are obtained using the approach developed by Nervana [22].

4.5 Experimental Evaluation of Prediction Accuracy

We next present results from experimental evaluation of prediction accuracy of abstract kernel emulation on two GPU systems: an Nvidia K20c with 13 Kepler SMs, 5GB global memory, 706MHz, 1.25MB L2 cache, and 48KB shared memory, and an Nvidia Titan X with 28 Pascal SMs, 12GB global memory, 1417MHz, 4MB L2 cache, and 96KB shared memory. We evaluate *all* the 58 kernels in the Rodinia benchmark suite. Those kernels collectively include 369 if-then-else conditions and loop bounds dependent on known parameters, 155 if-then-else conditions dependent on dynamically computed values, and 11 loops with statically unresolvable loop bounds.

Each kernel's binary was extracted and the SASS code [23] was subjected to abstract emulation, using the respective parameters for both targeted GPUs. The benchmarks were also executed to measure actual execution time. For each kernel in each benchmark, the prediction error, (pred-actual)/actual, is shown in Fig. 6. SAAKE models the time

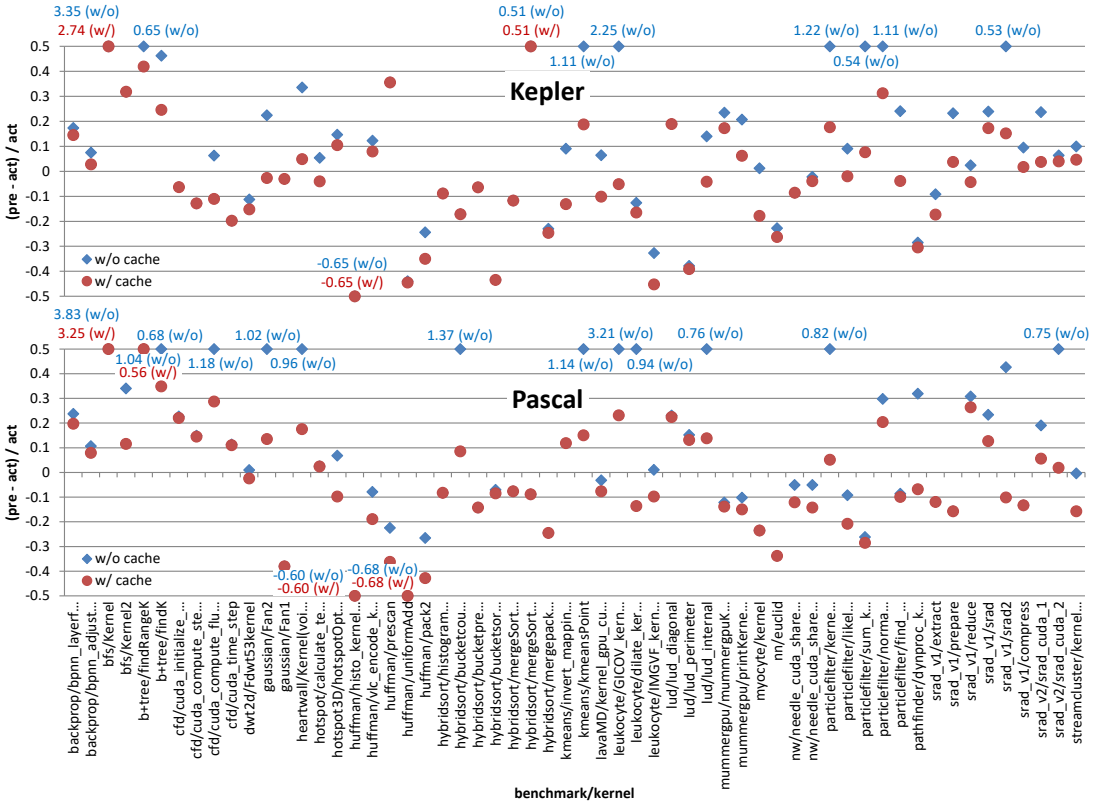

Fig. 6. Performance modeling accuracy with Rodinia benchmarks (top: Kepler, bottom: Pascal)

required for executing a maximally concurrently loadable set of thread blocks on one SM. This time is then scaled based on the actual number of thread blocks and number of SMs on the GPU to predict the entire kernel execution time.

Table 1. Geometric mean and median of error(=$\left|\frac{predicted-actual}{actual}\right|$) in Fig. 6

	Kepler		Pascal	
	w/o cache	w/ cache	w/o cache	w/ cache
(geo) mean	16.9%	11.8%	16.4%	13.7%
median	20.2%	13.8%	21.9%	13.2%

The plots show two sets of data, respectively with and without including L2 cache effect modeling described in subsection 4.4. As explained earlier, the L2 cache miss rate parameter can be obtained by incorporating a cache miss prediction model, or, as done here, by actual measurement. For both machines, the execution time is predicted quite

well for a majority of the kernels, especially when cache modeling is included. We note that gathering cache miss data from hardware counters to use in the emulator is well justified, since it is very quick and the ultimate use of the kernel emulation is not just to predict kernel execution time, but to identify hardware bottlenecks and use that information in code optimization, as presented in the following sections. We are unaware of any automatable GPU performance modeling approach that has demonstrated a comparable level of prediction accuracy across such a wide range of kernels. Aggregated accuracy metrics over the full set of kernels is presented in Table 1. The accuracy for many of the benchmarks can be further improved with enhancements to the emulator. Below we elaborate on the reasons for a relatively high error for some benchmarks and how some of those errors can be lowered.

SAAKE conservatively considers all branches of conditional expression to be executed. However, in kernels like BFS, only a few data dependent branches are executed. The effectiveness of coalescing is another factor that affects performance. For codes with indirect-memory accesses, the number of actual DRAM required transactions (effectiveness of coalescing) cannot be statically analyzed in general. The serialization effect of atomic operations on data-dependent memory locations cannot be estimated statically. Hence, SAAKE assumes that the atomic operations are done on different memory locations. In huffman/histo_kernel, the instruction "atomicAdd(&temp[buffer[i]], 1);" depends on the data, and the values of buffer[i] are largely 116 or 129. This results in very significant serialization overhead, which explains the low prediction accuracy. When the kernel has very few instructions, the achieved occupancy is lower than the computed occupancy (obtained using [6]) which affects the prediction accuracy. Further, if the total execution time of a kernel is less than 5 micro-seconds (e.g., huffman/uniformAdd), then kernel launch overhead dominates the kernel execution time and this affects prediction accuracy.

4.6 Limitations

The evaluation using Rodinia benchmark kernels shows good prediction accuracy for a large number of kernels, but also very high errors for some kernels. Modeling error can be high when data-dependent effects have a significant impact on execution time. This

is because the efficiency of the kernel emulation approach derives from the fact that actual simulation of all operations is not performed, but abstracted kernel emulation is performed for only a tiny fraction of all thread blocks of a GPU kernel. The following types of kernels can incur high modeling error:

- Kernels where a non-trivial fraction of global-memory accesses are actually returned from cached data in L2 cache: since the actual execution of instructions is not modeled, the best we can do is to randomly model a global memory access as returning from cache or global-memory based on estimated/known cache miss rates. Further, multi-level memory hierarchy is not modeled and can result in high error rates for some kernels.
- Kernels with significant thread divergence: Since conditional expressions in kernel code are not actually simulated, codes with significant data-dependent conditional execution can result in high modeling error.

Despite the above limitations to prediction accuracy, there are many practical scenarios where the abstract emulation approach is sufficiently accurate. An important use case is that of domain-specific code-generators that use tiling and effective use of shared memory. With such codes, data is explicitly copied from global memory to shared memory and reused multiple times from shared memory. Frequently repeated accesses to data in global-memory does not occur with such kernels and the prediction accuracy from abstract kernel emulation tends to be high. We demonstrate two such use-cases later in this paper.

5 MODEL-DRIVEN SEARCH AND ENHANCED AUTO-TUNING

In this section, we describe how a search directed by bottleneck-analysis with SAAKE can be coupled with auto-tuning systems like OpenTuner [1] that use ensemble search techniques and allow inclusion of custom search strategies.

5.1 Model-driven Search

The essential idea behind SAAKE's model-driven search is to start from an initial configuration and iteratively move along the canonical search-space directions by either doing

reshaping, thread coarsening, block coarsening, or unrolling, until a termination condition is reached. The decision on which direction to move is driven by the most significant bottleneck of the current configuration. Once the direction of a move is decided, the distance of a move (multiplicative factor) is the amount that enables maximum expected alleviation of the bottleneck (e.g., moving from GM latency-bound to SM throughput-bound). This distance is evaluated by extrapolating (linearly) the impact of the coalescing on the resource usage (reuses, concurrency).

Algorithm 2: Main Search Algorithm

```
1  prediction(c): predicted execution time for configuration c
2  occupancy(c): occupancy of configuration c
3  ustrides: set of all canonical vectors along which we want the configuration to evolve (multiplicative
     factors. Set to (2)^{3d+1} with d the grid dimension)
   input  : Original code
   output : Predicted best coarsening configuration
4  function entry()
5      curr = init_config
6      exec ← prediction(curr)
7      occu ← occupancy(curr)
8      (best_exec, best_curr) ← (exec, curr)
9      (prev_exec, prev_curr) ← (∞, ×)
10     while True do
11         curr′ ← next_best(curr, ustrides)
12         exec′ ← prediction(curr')
13         occu′ ← occupancy(curr′)
14         if occu′ < occu then
15             if best_exec ≥ prev_exec then
16                 break
17             (prev_exec, prev_curr) ← (best_exec, best_curr)
18             (best_exec, best_curr) ← (∞, ×)
19         if exec′ < best_exec then
20             (best_exec, best_curr) ← (exec′, curr′)
21         occu ← occu′
22     return prev_curr
```

The *utilization* of a resource, and the time *prediction* are computed using the abstract kernel emulation (Algorithm 1). Both the abstract emulation and the computation of the *occupancy* rely on NVCC to compile the current configuration.

Algorithm 3: Search Step Algorithm

$0 \leq U_{GM} \leq 1$: Global memory utilization factor
$0 \leq U_{SM} \leq 1$: Shared memory utilization factor
conf: current configuration
#T(c): number of threads in a thread-block for configuration c
#TB(c): number of thread blocks in an SM for c
maxT: hardware thread limit in an SM
occupancy(c): occupancy of c ($\#\mathrm{T}(c) \times \#\mathrm{TB}(c)/\mathrm{maxT}$)
olist: list of occupancies
utilization$_r(c)$**:** Utilisation factor of configuration c for resource r
transactions$_r(c)$**:** number of transactions to resource r in c
bottleneck$(c,\ o)$**:** bottleneck of configuration c with occupancy o (from Δ-analysis)
bott: main bottleneck of the current configuration
ustrides: set of all canonical vectors along which configuration may evolve (multiplicative factors)
s: canonical vector of multiplicative factor
K: scalar multiplicative factor
#regs: total number of registers per SM
maxregs: maximal number of registers per thread for the chosen occupancy

input : *conf*: Current configuration
ustrides
output: Predicted next best configuration

function ***next_best***(*conf*, *ustrides*)
  bott ← bottleneck(*conf*, occupancy(*conf*))
  if *bott.type* = *Latency* **then**
    $r \leftarrow$ *bott*.resource
    find $s \in$ *ustrides* **that maximizes** utilization$_r$(*conf* $\otimes$ s)
    $\Delta U_r \leftarrow$ utilization$_r$(*conf* $\otimes$ s) $- U_{GM}$
    find min K **s.t.** $U_{GM} + \Delta U_r(K-1) \geq 1$
    conf ← *conf* $\otimes$ $K.s$
  else `// throughput-limited`
    olist $\leftarrow \{k \times \#\mathrm{TB}(c)/\mathrm{maxT} \mid 1 \leq k < \#\mathrm{T}(c)\}$
    find biggest *occu* **s.t.** bottleneck(*conf*, *occu*) $\neq$ *bott*
    bott ← bottleneck(*conf*, *occu*)
    maxregs $\leftarrow \lfloor \#regs/(occu \times \mathrm{maxT}) \rfloor$
    configs ← gen_configs(*maxregs*, *occu*)
    $r \leftarrow$ *bott*.resource
    find *conf* $\in$ *configs* **that minimizes** transactions$_r$(*conf*)
  return *conf*

Alg 2 and Alg 3 describe the search algorithm used to traverse the configuration space. Alg 2 provides an overview of the algorithm. It uses Function *next_best* to find the best move (Algorithm 3). SASS code for the original baseline CUDA code is first passed to the bottleneck analyzer to predict the execution time (line 6). Based on the resource usage, the bottleneck analyzer also computes the achievable occupancy.

For bottleneck-guided traversal of the configuration space, the profitability of moving along each dimension is assessed. This is done by using unit stride analysis. Unit stride analysis is performed by moving a "unit" distance along each dimension of the configuration space, one at a time, starting from the initial configuration. For each such unit stride configuration, the global memory and shared memory traffic are estimated from abstract kernel emulation and saved in a table.

To navigate the configuration space, the search algorithm uses the bottleneck analyzer to predict the bottleneck and moves in a direction to alleviate the bottleneck. Given the current configuration, Alg 3 is used to predict the next configuration to be chosen.This step is explained in detail in the following paragraphs. For the chosen configuration, the occupancy and predicted execution time are estimated using the bottleneck analyzer. If the best predicted execution time among configurations evaluated by Alg 2 at the currently tested occupancy is lower than that at the previous occupancy, the best configuration at the current occupancy replaces any previous selection as the current best configuration seen so far and the traversal in that direction is continued. On the other hand, if the best configuration among all evaluated cases at the currently tested occupancy has a higher predicted time than the best recorded one at the previously tested occupancy, the search is terminated and the previously save best configuration is returned.

Given a configuration, Alg 3 describes the search algorithm to select the next configuration. The first step is to determine the current configuration's bottleneck using sensitivity analysis (line 2). If the current configuration is limited by a resource's latency, concurrency is increased to enhance latency tolerance. This is done by coarsening along a direction that will increase ILP for that resource. If there is data sharing across threads/blocks, coarsening helps reduce the total volume of traffic to the latency-bound resource. However, coarsening may increase the resources required per thread (registers,

shared-memory), possibly resulting in a reduction in warp-level parallelism. However, by coarsening, the increased ILP and reduced data traffic may help improve performance. The traversal direction is chosen to be the one with maximum reduction in data traffic for that particular resource. For this, results from unit stride analysis are used. The total stride of the move is chosen as the minimum number that would result in full utilization of the bottleneck resource (line 7).

On the other hand, if the resource is throughput-limited, the total number of transactions to that resource is reduced. This can be done by coarsening, if different threads/blocks share data. Since coarsening may reduce WLP, the amount of occupancy that can be sacrificed without dropping performance is determined. This is done by using abstract kernel emulation for the current configuration, for various occupancies lower than the current occupancy. The occupancy is lowered till the bottleneck changes. Then the amount of additional resources gained by lowering the occupancy is determined. The higher the coarsening factor, the better the reuse of shared data elements. The configuration that has the highest reduction in data volume is selected that can still achieve the minimum required occupancy.

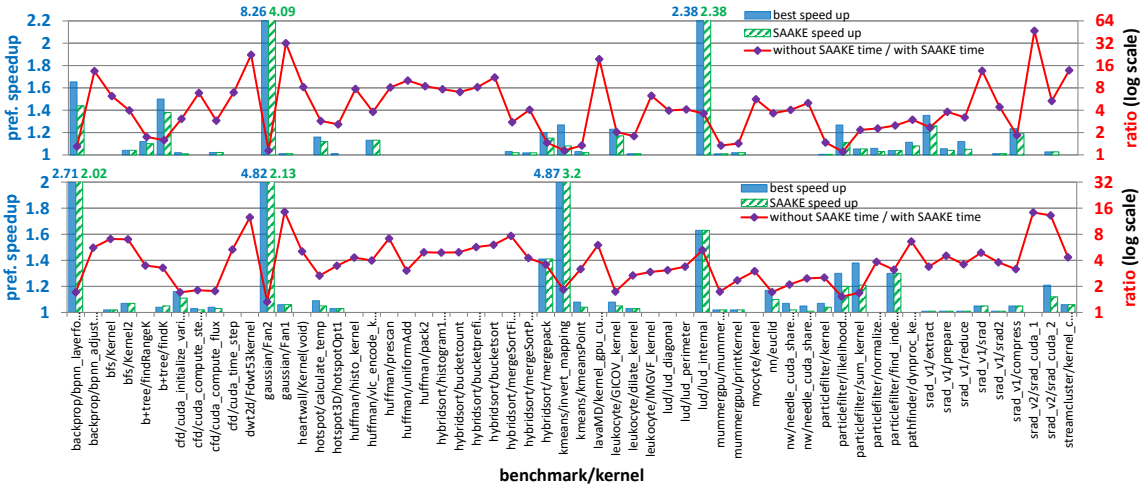

Fig. 7. OpenTuner Auto-tuning: All Rodinia Kernels

5.2 Coupling with Auto-tuning

OpenTuner is a general framework for auto-tuning which uses an ensemble of techniques to navigate the search space. Initially, a random seed configuration is determined and different search techniques are invoked. The resulting configurations are then run by OpenTuner to identify the best one. Each technique is then allotted a time slot which is proportional to the quality of the result produced by it. Mechanisms are provided to communicate through a common database to enable cooperation among different techniques. We coupled SAAKE with OpenTuner. In coupled-mode, SAAKE is invoked just before OpenTuner begins its initial search.

SAAKE starts with the base configuration (i.e., the initial code) and the predicted configuration is compiled (not run) to generate SASS code. The SASS code is then analyzed to determine the bottleneck and this is used to predict the next configuration. This process is repeated till SAAKE finds a configuration which is best according to its model-driven search. We note that no measured metrics like cache hit rates were provided to SAAKE for these experiments. The final configuration from SAAKE is used as the initial seed for OpenTuner. OpenTuner then proceeds with its ensemble of techniques to find the best configuration. In our experiments, in most cases, the output of SAAKE is either the optimal one or very close to the optimal, which significantly reduces the time required for OpenTuner to find the best configuration.

Table 2. CCSD(T) Tensor Contraction: SAAKE and/or OpenTuner

Kernel	Mac	Best Config	perf (GFLOP)			Tuning time					
						min		avg		max	
			Base	SAAKE	Best	w/o	w/	w/o	w/	w/o	w/
sd_t_d1_5	k	(16,16,2,3,1,1)	87	157	157	1091	47	1452	50	1702	56
	p	(32,8,2,4,1,1)	178	223	242	279	71	544	164	865	294
sd_t_d1_6	k	(16,8,2,2,1,1)	79	106	129	612	58	981	218	1359	679
	p	(16,8,2,1,1,4)	178	208	226	419	61	2143	150	4188	333

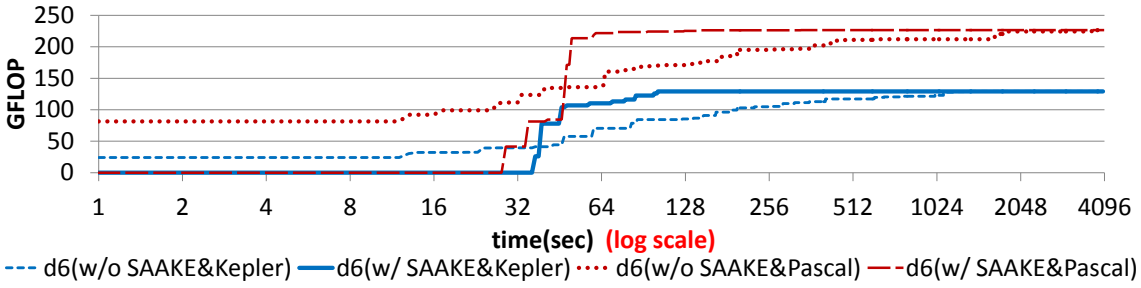

Fig. 8. OpenTuner Auto-tuning: Tensor Contractions

Table 3. Geometric mean and median in Fig. 7

	Kepler			Pascal		
	Best Speed up	SAAKE Speed up	Ratio of time w/o SAAKE vs w/ SAAKE	Best Speed up	SAAKE Speed up	Ratio of time w/o SAAKE vs w/ SAAKE
(geo) mean	1.11	1.09	4.16	1.13	1.09	3.72
median	1.01	1.01	3.88	1.02	1.01	3.53

5.3 Experimental Evaluation

We evaluated the effectiveness of the bottleneck guided optimization approach on the entire set of Rodinia benchmark kernels. We compare three scenarios: 1) SAAKE only, 2) OpenTuner only, and 3) SAAKE coupled with Opentuner. Fig. 7 presents the results for Kepler and Pascal GPUs. The vertical bars show the achieved speedup over the base Rodinia kernel (scale is linear, shown on the left). Each benchmark has a pair of bars: striped bar for SAAKE-only, and solid bar for SAAKE+OpenTuner. Roughly half of the kernels achieve some speedup by changing the original kernel. In about half of those, SAAKE by itself achieves the maximum possible performance improvement, and gets a good fraction of achievable speedup for most of the others. The connected lines in the two charts show the ratio of time taken by OpenTuner-Only versus the Coupled SAAKE+OpenTuner to find the best configuration. The scale for this data is logarithmic (scale on the right). It may be seen that SAAKE enables significant acceleration for OpenTuner by providing it a very good starting configuration. Results are summarized in Table 3.

Figure 8 and Table 2 present data for evaluation of SAAKE and OpenTuner for optimizing tensor contraction kernels. Tensor contractions are at the core of many computational chemistry models such as the coupled cluster methods [29]. Due to the significant fraction of compute time spent in performing tensor contractions, developers of the NWChem [32] computational chemistry suite created a domain-specific code generator [19] to synthesize efficient GPU kernels for tensor contractions. The NWChem suite includes a separate customized GPU kernel for each of 27 tensor contractions for the CCSD(T) method. These customized GPU kernels represent the current state-of-the-art one for this set of contractions [18, 19]. CUDA source code for these tensor contraction kernels is available in the open-source NWChem software distribution [24].

Three of the CCSD(T) tensor contractions are shown below.

```
sd_t_d1_1: T3(h3,h2,h1,p6,p5,p4) -= t2(h7,p4,p5,h1)*v2(h3,h2,p6,h7)
 [...]
sd_t_d1_5: T3(h3,h1,h2,p5,p4,p6) += t2(h7,p4,p5,h1)*v2(h3,h2,p6,h7)
sd_t_d1_6: T3(h1,h3,h2,p5,p4,p6) -= t2(h7,p4,p5,h1)*v2(h3,h2,p6,h7)
 [...]
```

Table 2 presents performance data for two of the CCSD(T) tensor contractions. For each contraction, and each of the two machines, the best configuration found by SAAKE+OpenTuner is shown, along with the achieved performance for the base version, SAAKE-Only, and SAAKE+OpenTuner. It may be seen that SAAKE-Only achieves a high fraction of the speedup achieved by SAAKE+OpenTuner. The time for performing auto-tuning with-/without SAAKE is shown in terms of min/max/average. It may be seen that integrating OpenTuner with SAAKE results in significant reduction in the average tuning time, as well as the maximum time, which can be over an hour (4188 seconds). Fig. 8 shows the trajectory over time for OpenTuner versus SAAKE+OpenTuner for one of the benchmarks. It may be seen that for both machines, about a 10x decrease in tuning time is achieved.

6 ASSISTING MANUAL OPTIMIZATION: CASE STUDIES

As discussed in Sec. 5, SAAKE can be used to identify the resource bottleneck(s) for a given GPU kernel. We demonstrate through two detailed case-studies the bottleneck insights from sensitivity analysis can provide extremely useful information to complement information obtainable through performance tools such as NVPROF and NSIGHT. On the one hand, performance tools like NVPROF and NSIGHT provide very accurate information based on actually measured hardware counter data, while SAAKE bottleneck analysis is based on a simple approximate model of the execution. But on the other hand, SAAKE's sensitivity analysis enables the kind of specific "what-if" exploration with respect to critical resource parameters that is not feasible with the more accurate measurement-based tools.

We present the following illustrative case studies: i) further improving the performance of the tensor contractions discussed in the previous section, and ii) improving code for stencil computations that was synthesized by a state-of-the-art stencil code generator for GPUs [27].

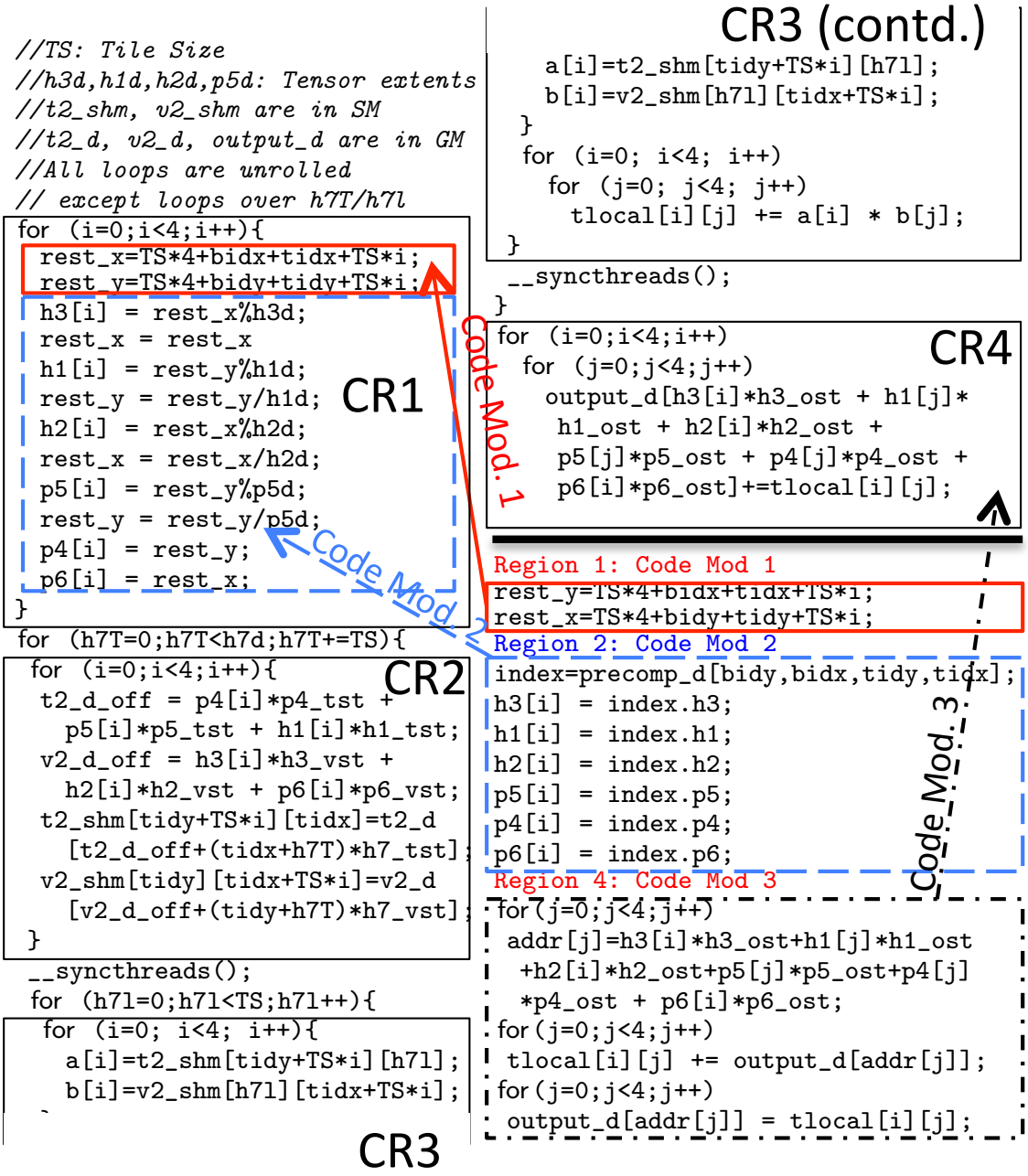

Fig. 9. SAAKE-based optimization of tensor contraction kernel

Table 4. SAAKE optimization steps for sd-t-d1-6 on Kepler

Orig. Code (92 GFLOPs)

	total	CR 1	CR 2	CR 3	CR 4
GM LAT	3%	0%	1%	0%	2%
SFU LAT	16%	16%	0%	0%	0%
GM THR	58%	0%	6%	0%	52%
SM THR	7%	0%	0%	7%	0%

(a)

Code Mod. 1 (123 GFLOPs)

	total	CR 1	CR 2	CR 3	CR 4
GM LAT	38%	0%	1%	0%	35%
SFU LAT	19%	19%	0%	0%	0%
GM THR	22%	0%	11%	0%	7%
SM THR	8%	0%	0%	8%	0%

(b)

Code Mod. 2 (139 GFLOPs)

	total	CR 1	CR 2	CR 3	CR 4
GM LAT	41%	0%	2%	0%	40%
SFU LAT	0%	0%	0%	0%	0%
GM THR	31%	0%	14%	0%	13%
SM THR	9%	0%	0%	9%	0%

(c)

Code Mod. 3 (161 GFLOPs)

	total	CR 1	CR 2	CR 3	CR 4
GM LAT	11%	0%	2%	0%	10%
SFU LAT	0%	0%	0%	0%	0%
GM THR	59%	0%	16%	0%	39%
SM THR	11%	0%	0%	11%	0%

(d)

Table 5. SAAKE optimization steps for sd-t-d1-6 on Pascal

Orig. Code (166 GFLOPs)

	total	CR 1	CR 2	CR 3	CR 4
GM LAT	9%	0%	6%	0%	3%
GM THR	48%	0%	0%	0%	48%
INT THR	6%	6%	0%	0%	0%
DP THR	32%	0%	0%	32%	0%

Code Mod. 1 (190 GFLOPs)

	total	CR 1	CR 2	CR 3	CR 4
GM LAT	32%	0%	25%	0%	7%
GM THR	13%	0%	0%	0%	13%
INT THR	9%	9%	0%	0%	0%
DP THR	38%	0%	0%	38%	0%

Code Mod. 2 (225 GFLOPs)

	total	CR 1	CR 2	CR 3	CR 4
GM LAT	21%	0%	13%	0%	8%
GM THR	14%	0%	0%	0%	14%
INT THR	12%	10%	2%	0%	0%
DP THR	46%	0%	0%	46%	0%

Code Mod. 3 (250 GFLOPs)

	total	CR 1	CR 2	CR 3	CR 4
GM LAT	3%	0%	3%	0	1%
GM THR	23%	0%	0%	0	23%
INT THR	14%	11%	3%	0	0%
DP THR	53%	0%	53%	0	0%

6.1 Tensor Contraction Kernel

We present the use of SAAKE in optimizing the sd_t_d1_6 tensor contraction kernel, one of the more complex cases to optimize. The same process can be applied to any of the CCSD(T) kernels. Fig. 9 shows the structure of the current code in NWChem's GPU kernel as well as the sequence of changes made during the optimization process. SAAKE analysis of the kernel code revealed sensitivity to multiple hardware resources. A hierarchical strategy was used to partition code regions: first run a coarse analysis without partitioning and then refine based on results of the analysis, drilling down into smaller regions that are the focus of sensitivity analysis. For this kernel, analysis resulted in partitioning the kernel code into four code regions CR1, CR2, CR3, and CR4, demarcated in Fig. 9. The sequence of code modifications (explained later) is also shown in the latter portion of the figure, with arrows connecting the modified code with the original code. Each thread computes a data slice of the result tensor $output\_d$ using needed elements from input tensors $t2\_d$ and $v2\_d$. Arrays with a "d" suffix denote data in "device" global memory, and those with suffix "shm" are in shared memory. CR1 computes base indices (for $h1, h2, h3, p4, p5, p6$) of the hyper-rectangular data slices of the tensors that

each thread operates upon. CR3 is the compute loop for the contraction over the index $h7$. In CR2, slices of the input tensors ($t2\_dandv2\_d$) are moved into shared-memory arrays. CR3 performs the floating-point operations to accumulate the results computed in a local array ($tlocal$, which is placed in registers by the compiler). CR4 writes out the final results to global memory ($output\_d$). The i/j loops in the pseudocode are fully unrolled in the actual CUDA kernel code, but are shown as loops for compactness of the pseudocode.

Table 4 shows results from sensitivity analysis of the original kernel code for a subset of resources that exhibited sensitivity: global memory (GM) latency, special function unit (SFU) latency, GM throughput (gap), and shared memory (SM) throughput. The left column of Table 4(a) shows the overall sensitivity for these resources, while the remaining four columns present the region-wise sensitivity for CR1, CR2, CR3, and CR4. The highest sensitivity is w.r.t. GM throughput and CR4 is the primarily affected code region. Index $h3$ is mapped to $threadIdx.x$ to achieve coalesced memory access in CR2. The fastest varying index (FVI) of the output array is $h1$. Since it is mapped to $threadIdx.y$, global memory accesses for accumulating results are uncoalesced. If $h1$ is mapped to $threadIdx.x$ and $h3$ to $threadIdx.y$, coalesced accesses can be achieved in CR4, while uncoalesced memory accesses would now occur in CR2. This swap is shown under *Code Mod. 1* in the pseudocode.

The results of applying SAAKE to the modified code are shown in Table 4(b). The performance increased from 92 GFLOPs for the original code to 123 GFLOPs and the sensitivity to global-memory throughput (GM-THR) in CR4 has dramatically reduced. There is now sensitivity to GM-THR in CR3, because the transposed mapping now causes uncoalesced reads in CR3. However, it is only 11%, compared to 52% for CR4. We note that the percentage change in time reflected by the sensitivity data is for the total kernel execution time and not just for the time attributable to each local region. We observe a 19% sensitivity to SFU latency. This is due to the modulo and division operations transformed into a sequence of operations including reciprocal that is executed by the SFU—this leads to many chains of dependences and, therefore, insufficient concurrency to tolerate the large SFU latency.

Since sensitivity to GM-THR for CR1 is zero (no GM loads/stores occur here), we alleviated the SFU bottleneck by precomputing the indices, storing them in global memory, and reading them from GM, instead of computing them. This is shown in Fig. 9 as *Code Mod. 2*.

After this optimization, performance increased to 139 GFLOPs. Table 4(c) shows the SAAKE results for this code version. Next, we sought to alleviate the GM-LAT bottleneck in CR4 (40% sensitivity), a consequence of inadequate concurrency to tolerate the long latency of the chained address-computation, read from GM and write to GM in CR4. *Code Mod. 3* shows split code for the same computation, where a set of independent address computations are first performed, followed by a set of GM reads to accumulate results in registers, followed by a set of GM writes. After this code change, performance increased to 161 GFLOPs and the GM-LAT sensitivity of CR4 decreases significantly from 40% to 10%. At this point, the kernel is limited by GM-THR on the output. Since each output element is only written out once and the GM stores are coalesced, no further optimization is attempted.

Table 5 shows the data for applying SAAKE to the same kernel on the Pascal GPU. The sensitivity metrics for the original code are quite different from those seen on Kepler. A significant reason is that this Pascal P102 GPU has very low double-precision performance (peak of 343 GFLOPs). Due to space limitations, we do not provide details on the sequence of code modifications. However, the sensitivity analysis metrics in Table 5 demonstrate significant differences.

Table 6 compares the performance of the original kernel with the two optimized versions on both GPUs. The Kepler-optimized version achieves the lower performance of 219 GFLOPs on the Pascal, compared to 250 GFLOPs for the Pascal-optimized kernel, but it is better than the original kernel's 166 GFLOPs. But the Pascal-tuned kernel only achieves 49 GFLOPs on the Kepler GPU, as compared to 92 GFLOPS for the original kernel and 161 GFLOPs for the Kepler-optimized version. The reason is as follows: In the optimized code for the Pascal system, shared memory operations were changed to global memory instructions to utilize the Double Precision unit efficiently. Hence, this

Table 6. Performance of optimized kernels on Kepler and Pascal GPUs

Mode	Kepler GFLOPs	Pascal GFLOPs
Original	92	166
Kepler-Opt	161 (1.75x speedup)	219 (1.32x speedup)
Pascal-Opt	49 (0.53x speedup)	250 (1.51x speedup)

code causes more global memory transactions. However, this strategy is very detrimental on the Kepler machine which has a lower memory bandwidth and many more double precision units.

Similar SAAKE analysis and optimization was carried out for some other tensor contraction kernels in the set. Based on the gained insights, the tensor contraction code generator that produced the codes in NWChem was modified so that the emitted code structure was like the optimized versions that had been manually generated through SAAKE-based optimization. Fig. 10 charts the performance on 18 tensor contraction kernels used in the NWChem CCSD(T) method. Experiments were conducted for three tensor sizes: all tensor extents of 16, all-15, and all-17. We observe that the modified code generator generates kernels that execute consistently faster than the ones currently used in the NWChem distribution. Results are summarized in Table 7.

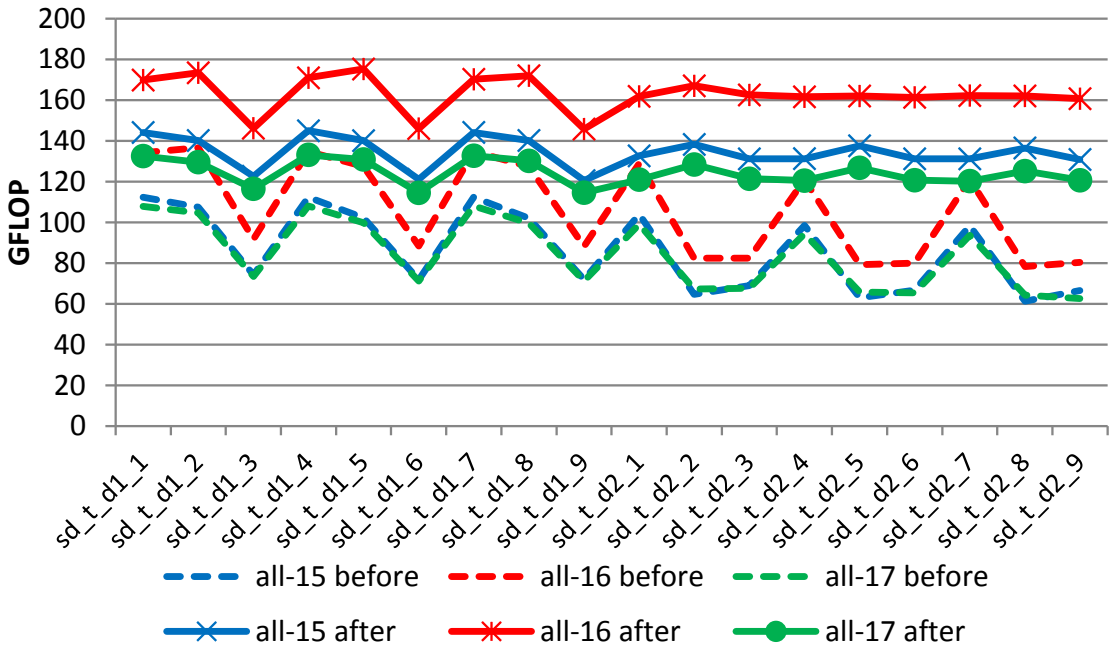

Fig. 10. Performance of original kernels versus new kernels from modified code generator

Table 7. Geometric mean and median of GFLOPs in Fig. 10

	all-15 before	all-15 after	all-16 before	all-16 after	all-17 before	all-17 after
(geo) mean	84.3	134.2	103.9	162.6	82.9	124.3
median	86.3	134.6	106.3	162.2	83.6	123.3

6.2 Optimizing the Hypterm Stencil Computation

We carried out the second exercise on optimizing the Hypterm function from the Exp_CNS benchmark[2]. Due to space limitations, we only present a short summary here; full details are presented in a technical report [12].

The Hypterm function involves a sequence of 15 stencils, S_1, .. S_{15}. Optimized GPU code generation for the Hypterm function was presented as a case study by Rawat et al. for a domain-specific compiler for optimizing stencil computations on GPUs [27]. Groups of three adjacent stencils make contributions to the same output array and benefit from fusion to reduce data movement. Our starting point was a slight variant of the final fused configuration reported by Rawat et al. [27] (there were seven groups in that code, but we started with the following configuration of six kernels, where two singleton groups for S_{12} and S_{15} were fused to form a single group G6).

G1=$\{S_1, S_2, S_{13}, S_{14}\}$, G2=$\{S_3, S_{10}, S_{11}\}$, G3=$\{S_4, S_5, S_7, S_8\}$, G4=$\{S_6\}$, G5=$\{S_9\}$, and G6=$\{S_{12}, S_{15}\}$.

Analysis using SAAKE revealed that the six kernels were not latency-limited in terms of GM and SM data movements. Hence, we explored the further fusing of kernels to reduce data movement, at the cost of lowered occupancy, since the kernel was throughput-limited and not yet latency-limited for shared and global memory accesses. A fused kernel can decrease occupancy but increase ILP. We made 3 fused kernels from the 6 kernels, based on usage of the same inputs/outputs, reducing occupancy to 0.5 (by decreasing occupancy from 1 to 0.5, each thread/block was able to use more resources): G1'=G1,G2, G2'=G3,G4,G5, G3'=G6. Note that we reduce occupancy of the kernel G6 to eliminate register spills which may cause global memory movements. The transformation improved performance from 141 GFLOPs to 161 GFLOPs. SAAKE analysis on the three fused kernels revealed them to still be throughput limited, suggesting further fusion. So, we fused the 3 kernels to make one fused kernel, decreasing occupancy to 0.25, but improving

[2]https://ccse.lbl.gov/ExaCT/CNS_Nospec.tgz

performance. We also changed the thread-block sizes and tile sizes. The final achieved performance was 260 GFLOPs on the Kepler GPU, a significant improvement over the 141 GFLOPs of the initial version.

7 RELATED WORK

GPU Performance Modeling: Several studies have developed analytical approaches for modeling kernel performance on GPUs. The work of Hong et al. [13] represents one of the earliest efforts at modeling GPU performance in terms of the impact of warp level concurrency for computational operations and memory operations on performance. Sim et al. [30] extended the modeling approach of Hong et al. [13] to address more hardware features, and to provide feedback to application developers on four metrics that could guide them in identifying execution bottlenecks and how they may be overcome to improve the performance. Baghsorkhi et al. [2] developed an analytical approach to predicting GPU kernel performance by using a more detailed modeling of the assembly level statements of the kernel binary and inter-statement dependences. Zhang et al. [41] also developed a quantitative performance analysis model aimed at identifying the primary resource bottleneck for a GPU kernel. Lee et al. [16] proposed a DSL for performance modeling along with an analytical modeling framework, using static analysis of the program to estimate the number of floating point operations and memory accesses. Recently, Xu et al. [39] developed a precise analytical performance model for a cache-less heterogeneous many-core processor SW26010, the current top supercomputer [8]. The performance modeling approach extends that of Hong et al. [13], and its use in static performance tuning of a number of Rodinia benchmarks is demonstrated. Very high accuracy (average error of 5%) and very significant speedup (43x) over auto-tuning were achieved with high tuning quality (6% loss). Zhou et al. recently developed a performance analysis framework for Nvidia GPUs, with a focus on deep neural networks (DNNs) [42]. Using assumptions on the behavior of GEMM-like computations in DNNs, analytical models for potential resource bottlenecks are presented. These models take as parameters the result of assembly code (SASS) analysis to describe the input application.

The popular Roofline model [38] has applied in the context of GPU kernel optimization. The GPURoofline [14] system helps non-expert users tune GPU kernels for high performance. The roofline model is used to first identify the performance bottleneck, which is then relieved through iterative improvement via specific optimization techniques such as Reducing Dynamic Instructions (RDIS) and increasing Instruction-level Parallelism (ILP).

A notable difference between previously reported approaches to GPU performance modeling and the approach developed in this paper is that, to the best of our knowledge, none of the previous approaches has demonstrated applicability for *automated* performance modeling of arbitrary GPU kernels. We have demonstrated the use of the developed approach to modeling and code transformation on the entire set of Rodinia benchmarks, as well as non-trivial tensor-contraction benchmarks used in the NWChem computational chemistry suite.

Compiler Optimization for GPUs: Several research efforts have focused on compiler optimization for GPUs. Many of these efforts have been in the context of affine programs, where precise dependence analysis is feasible. Baskaran et al. [3] developed a C-to-CUDA transformation in the Pluto polyhedral optimizer [4]. The PPCG compiler [33] is another powerful polyhedral compiler for GPU code generation from C input program. It implements a variety of optimizations such as shared memory promotion. However, its cost models to determine transformation profitability remain very basic.

OpenACC [37] is a directive-based programming model for GPU computing, where a C program annotated with OpenACC directives is automatically transformed by the OpenACC compiler for execution on GPUs. OpenMP [7] also now offers an "Offload" mode that allows user-annotated C programs to be executed on GPUs, in a similar manner that OpenACC introduced several years ago. OpenARC [17] is a directive-based compiler that is intended to accept either OpenACC or OpenMP-Offload programs, and generates architecture-specific (including GPU targets) codes.

We did perform several tests with PPCG and OpenACC / OpenMP-Offload for tensor contraction examples, and observed that the achieved performance was considerably lower than that we were able to achieve in this paper. We believe our tools can be used

to improve performance of code generated by these compilers, by enabling the selection of better transformations if suitable interfaces are developed. But that may require significant implementation efforts.

Several efforts have developed Domain-Specific Languages and compilers for GPUs, e.g. [5, 9–11, 19, 26]. We demonstrated the use of our developed tools/methodology on two DSL examples in this paper. We believe that other DSLs for GPUs can benefit form them.

For thread coarsening in particular, Unkule et al. [31] developed an approach to analyze GPU code and perform source-level transformations to obtain GPU kernels with varying thread granularity. Magni et al. [20, 21] developed a thread coarsening tool for OpenCL GPU kernels, in conjunction with their research on developing a machine learning model for identifying the best thread coarsening factor. It would be interesting to compare the effectiveness of their approach with ours, but we were unable to do so since their software is not publicly available.

8 DISCUSSION AND CONCLUSION

Performance optimization requires a clear understanding of the impact of various program transformations on its execution time. In this paper we have presented a new approach to capture the key architectural features and their impact on application performance. The usefulness of the abstract kernel emulation approach was demonstrated, and also coupled with the OpenTuner auto-tuning framework.

In addition to the case studies presented, our approach can provide feedback to application developers in helping them identify potentially beneficial transformations. Because the approach only needs *latency* and *gap* parameters for key GPU resources, and not the availability of the actual hardware, it can also be helpful with codesign of algorithms and architectures to maximize performance for a specific application workload on a future system.

ACKNOWLEDGMENTS

We thank the reviewers of the paper for valuable feedback that helped improve the paper. This work was supported in part by the National Science Foundation through

awards 1440749, 1513120 and 1629548, and by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under award number 63823. Pacific Northwest National Laboratory is operated by Battelle for DOE under Contract DE-AC05-76RL01830. We are grateful to the Ohio Supercomputer Center for the use of computing resources.

REFERENCES

[1] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An Extensible Framework for Program Autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT '14)*. ACM, New York, NY, USA, 303–316. https://doi.org/10.1145/2628071.2628092

[2] Sara S Baghsorkhi, Matthieu Delahaye, Sanjay J Patel, William D Gropp, and Wen-mei W Hwu. 2010. An adaptive performance modeling tool for GPU architectures. In *ACM Sigplan Notices*, Vol. 45. ACM, 105–114.

[3] Muthu Baskaran, Jj Ramanujam, and P Sadayappan. 2010. Automatic C-to-CUDA code generation for affine programs. In *Compiler Construction*. Springer, 244–263.

[4] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *ACM SIGPLAN Notices*, Vol. 43. ACM, 101–113.

[5] Matthias Christen, Olaf Schenk, and Helmar Burkhart. 2011. PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS '11)*. IEEE Computer Society, 676–687.

[6] CUDA occupancy calculator [n. d.]. CUDA occupancy calculator. https://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

[7] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5, 1 (1998), 46–55.

[8] Jack Dongarra. 2016. Report on the sunway taihulight system. *PDF). www. netlib. org. Retrieved June* 20 (2016).

[9] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. 2014. Hybrid Hexagonal/Classical Tiling for GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. ACM, Article 66, 10 pages.

[10] Tobias Grosser, Albert Cohen, Paul H. J. Kelly, J. Ramanujam, P. Sadayappan, and Sven Verdoolaege. 2013. Split Tiling for GPUs: Automatic Parallelization Using Trapezoidal Tiles. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units (GPGPU-6)*. ACM, 24–31.

[11] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. 2012. High-performance Code Generation for Stencil Computations on GPU Architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS '12)*. ACM, 311–320.

[12] Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, Prashant Singh Rawat, Sriram Krishnamoorthy, Louis-Noel Pouchet, Fabrice Rastello, and P. Sadayappan. 2018. *GPU Code Optimization using Abstract Kernel Emulation and Sensitivity Analysis*. Technical Report. Ohio State University.

[13] Sunpyo Hong and Hyesoon Kim. 2009. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *ACM SIGARCH Computer Architecture News*, Vol. 37. ACM, 152–163.

[14] Haipeng Jia, Yunquan Zhang, Guoping Long, Jianliang Xu, Shengen Yan, and Yan Li. 2012. GPURoofline: a model for guiding performance optimizations on GPUs. In *European Conference on Parallel Processing*. Springer, 920–932.

[15] Junjie Lai and André Seznec. 2013. Performance upper bound analysis and optimization of SGEMM on Fermi and Kepler GPUs. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*. IEEE, 1–10.

[16] Seyong Lee, Jeremy S. Meredith, and Jeffrey S. Vetter. 2015. COMPASS: A Framework for Automated Performance Modeling and Prediction. In *ACM International Conference on Supercomputing (ICS15)*. https://doi.org/10.1145/2751205.2751220

[17] Seyong Lee and Jeffrey S Vetter. 2014. OpenARC: Open Accelerator Research Compiler for Directive-Based, Efficient Heterogeneous Computing. In *HPDC '14: Proceedings of the ACM Symposium on High-Performance Parallel and Distributed Computing, Short Paper*. https://doi.org/10.1145/2600212.2600704

[18] Wenjing Ma, Sriram Krishnamoorthy, Oreste Villa, and Karol Kowalski. 2010. Acceleration of Streamed Tensor Contraction Expressions on GPGPU-Based Clusters. In *Proceedings of the 2010 IEEE International Conference on Cluster Computing, Heraklion, Crete, Greece, 20-24 September, 2010*. 207–216. https://doi.org/10.1109/CLUSTER.2010.26

[19] Wenjing Ma, Sriram Krishnamoorthy, Oreste Villa, Karol Kowalski, and Gagan Agrawal. 2013. Optimizing tensor contraction expressions for hybrid CPU-GPU execution. *Cluster computing* 16, 1 (2013), 131–155.

[20] Alberto Magni, Christophe Dubach, and Michael O'Boyle. 2014. Automatic optimization of thread-coarsening for graphics processors. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 455–466.

[21] Alberto Magni, Christophe Dubach, and Michael FP O'Boyle. 2013. A large-scale cross-architecture evaluation of thread-coarsening. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 11.

[22] Nervana maxas [n. d.]. Nervana maxas. https://github.com/NervanaSystems/maxas/

[23] NVIDIA SASS 2018. CUDA Binary Utilities. http://docs.nvidia.com/cuda/cuda-binary-utilities/index.html

[24] NWChem Download 2017. NWChem Download. http://www.nwchem-sw.org/index.php/Download

[25] Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Henry Wong. 2009. Micro-benchmarking the GT200 GPU. *Computer Group, ECE, University of Toronto, Tech. Rep* (2009).

[26] Mahesh Ravishankar, Paulius Micikevicius, and Vinod Grover. 2015. Fusing Convolution Kernels Through Tiling. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY 2015)*. ACM, 43–48.

[27] Prashant Rawat, Changwan Hong, Mahesh Ravishankar, Vinod Grover, Louis-Noel Pouchet, Atanas Rountev, and P. Sadayappan. 2016. Resource Conscious Reuse-Driven Tiling for GPUs. In *International Conference on Parallel Architectures and Compilation Techniques*. 99–111.

[28] Timothy G Rogers, Mike O'Connor, and Tor M Aamodt. 2012. Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 72–83.

[29] Isaiah Shavitt and Rodney J Bartlett. 2009. *Many-body methods in chemistry and physics: MBPT and coupled-cluster theory*. Cambridge university press.

[30] Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim, and Richard Vuduc. 2012. A performance analysis framework for identifying potential benefits in GPGPU applications. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 11–22.

[31] Swapneela Unkule, Christopher Shaltz, and Apan Qasem. 2012. Automatic restructuring of GPU kernels for exploiting inter-thread data locality. In *International Conference on Compiler Construction*. Springer, 21–40.

[32] Marat Valiev, Eric J Bylaska, Niranjan Govind, Karol Kowalski, Tjerk P Straatsma, Hubertus JJ Van Dam, Dunyou Wang, Jarek Nieplocha, Edoardo Apra, Theresa L Windus, et al. 2010. NWChem: a comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications* 181, 9 (2010), 1477–1489.

[33] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013), 54.

[34] Mark N Wegman and F Kenneth Zadeck. 1991. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 2 (1991), 181–210.

[35] Whitepaper 2012. NVIDIA Tesla K100. http://www.nvidia.com/content/PDF/product-specifications/GeForce_GTX_680_Whitepaper_FINAL.pdf

[36] Whitepaper 2016. NVIDIA Tesla P100. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

[37] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. 2012. OpenACC-first experiences with real-world applications. *Euro-Par 2012 Parallel Processing* (2012), 859–870.

[38] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[39] Shizhen Xu, Yuanchao Xu, Wei Xue, Xipeng Shen, Xiaomeng Huang, and Guangwen Yang. 2018. Taming the "Monster": Overcoming program optimization challenges on SW26010 through precise performance modeling. In *Parallel and Distributed Processing Symposium (IPDPS), 2018 IEEE International*. IEEE, pages will be added.

[40] Xiuxia Zhang, Guangming Tan, Shuangbai Xue, Jiajia Li, Keren Zhou, and Mingyu Chen. 2017. Understanding the GPU Microarchitecture to Achieve Bare-Metal Performance Tuning. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 31–43.

[41] Yao Zhang and John D Owens. 2011. A quantitative performance analysis model for GPU architectures. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE, 382–393.

[42] Keren Zhou, Guangming Tan, Xiuxia Zhang, Chaowei Wang, and Ninghui Sun. 2017. A performance analysis framework for exploiting GPU microarchitectural capability. In *Proceedings of the International Conference on Supercomputing*. ACM, 15.